The AArch64 processor (aka arm64), part 16: Conditional execution

devblogs.microsoft.com/oldnewthing/20220817-00

August 17, 2022

Raymond Chen

The AArch64 provides a handful of branchless conditional instructions.

First up are the conditional assignments.

```
; condition select
; Rd = cond ? Rn : Rm
csel    Rd/zr, Rn/zr, Rm/zr, cond

; conditional select invert
; Rd = cond ? Rn : ~Rm
csinv   Rd/zr, Rn/zr, Rm/zr, cond

; conditional select negate
; Rd = cond ? Rn : -Rm
csneg   Rd/zr, Rn/zr, Rm/zr, cond

; conditional select increment
; Rd = cond ? Rn : (Rm + 1)
csinc   Rd/zr, Rn/zr, Rm/zr, cond
```

These operations assign a value based on a condition. If the condition is met, then the first input operand is assigned to the destination. Otherwise, some function of the second input operand is assigned.

The condition is any of the same condition codes used by the conditional branch instruction.

By passing the same register as both input operands, you get some interesting pseudo-instructions:

```
; conditional invert
; Rd = cond ? Rn : ~Rn
cinv    Rd/zr, Rn/zr, cond  ; csinv Rd, Rn, Rn, cond

; conditional increment
; Rd = cond ? (Rn + 1) : Rn
cinc    Rd/zr, Rn/zr, cond  ; csinc Rd, Rn, Rn, !cond

; conditional negate
; Rd = cond ? Rn : -Rn
cneg    Rd/zr, Rn/zr, cond  ; csneg Rd, Rn, Rn, !cond
```

Since the interesting operation occurs to the second input operand, we have to reverse the sense of the condition. (The assembler doesn't accept `!` to negate the condition. You'll have to write it out by hand.)

Finally, we get some interesting pseudo-instructions if we hard-code both input registers to zero.

```
; conditional set
; Rd = cond ? 1 : 0
cset    Rd/zr, cond         ; csinc Rd, zr, zr, !cond

; conditional set mask
; Rd = cond ? -1 : 0        ; -1 is all bits set
csetm   Rd/zr, cond         ; csinv Rd, zr, zr, !cond
```

The next set of conditional operations is the conditional comparisons, which let you combine the results of multiple comparisons so you can perform a single test at the end.

Recall that Itanium accomplished this by predicating a comparison instruction, which had the effect of accumulating (either by AND or OR) multiple predicates into a single predicate register. And PowerPC did this by having eight sets of flags on which you can perform boolean operations, so that you can combine the flags in the way you like to produce a single result bit at the end.

AArch64 does it by letting you make a comparison instruction conditional and also specify the artificial result if the condition is not met.

```
; conditional compare
; if (cond) then set flags as if "cmp a, b"
;           else set flags to #nzcv
ccmp    Rd/zr, #imm5, #nzcv, cond
ccmp    Rd/zr, Rn/zr, #nzcv, cond

; conditional compare negative
; if (cond) then set flags as if "cmn a, b"
;           else set flags to #nzcv
ccmn    Rd/zr, #imm5, #nzcv, cond
ccmn    Rd/zr, Rn/zr, #nzcv, cond
```

The immediate is an unsigned 5-bit value, so it can cover the range 0 … 31.

If the condition is met, then the flags are set according to the underlying comparison instruction. And if the condition is not met, then the flags are set to the bits you specify. The flags are expressed as a 4-bit value, corresponding to this arrangement of the flag bits:

N	Z	C	V

The pattern for combining two results via AND is

```
; branch if a1 op1 b1 && a2 op2 b2

cmp     a1, b1
ccmp    a2, b2, #op2-fail, op1
bop2    both_true
```

You start with the first comparison. Then you follow up with a `CCMP` where the condition is the thing you want the first comparison to be. The register operands are the arguments to the second comparison. And the `nzvc` value is chosen so that it fails the `Bop2` .

For example,

```
; branch if r0 ge 0 and r1 lt 5
cmp     r0, #0
ccmp    r1, #5, #0, ge
blt     both_true
```

Let's walk through this code. The important aspect of the magic value `#0` is that it corresponds to $N = 0$ and $V = 0$, which is the flags result of a comparison that reports "greater than or equal to". (You can consult the condition chart from last time to see what each condition tests.)

Instruction	Flags

	If *r0 ≥ 0*	If *r0 < 0*
`cmp r0, #0`	GE	LT
`ccmp r1, #5, #0, ge`	`cmp r1, #5`	GE

If the first comparison results in `GE`, then we perform the second comparison, and if it results in `LT` then we branch, satisfied that both conditions were met.

If the first comparison does not produce `GE`, then we force the *nzcv* to zero, which acts like `GE`, and do not perform the second comparison. We just force it to fail. The branch fails, because we forced the flags to the opposite of `LT`.

Similarly, the pattern for combining two comparisons via OR is

```
; branch if a1 op1 b1 && a2 op2 b2

cmp     a1, b1
ccmp    a2, b2, #op2-succeed, !op1
bop2    either_true
```

If the first comparison is not the desired `op1`, then we try again with the second comparison. But if the first comparison was what we wanted, then we force the flags to be something that causes the conditional branch to succeed.

This strikes me as a clever solution for allowing multiple conditions to be combined and tested with a single conditional branch at the end, and therefore consume only a single branch prediction slot. It gives you the results in a single flags register, rather than having to create multiple flags registers or predicates and then invent instructions that combine them. It works only for straight-line expressions (not things like `(a && b) || (c && d)`), but that's probably good enough.

Bonus chatter: The Windows debugger disassembles these instructions differently from how they are listed in the ARM reference manual. Instead of putting the condition at the end of the instruction, the condition is appended to the opcode.

```
csel    w0, w8, wzr, eq     ; ARM reference manual
cseleq  w0, w8, wzr         ; Windows debugger

ccmp    x0, #0x1c, #0, le   ; ARM reference manual
ccmple  x0, #0x1c, #0       ; Windows debugger
```

Raymond Chen

Follow